EFFICIENT FPGA ACCELERATION OF CONVOLUTIONAL DEEP NEURAL NETWORKS

Atul Rahman

Computer Engineering Program
Graduate school of UNIST

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Efficient FPGA Acceleration of Convolutional Deep Neural Networks

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Atul Rahman

06.14.2016
Approved by

[illegible]

Major Advisor
Jongeun Lee

Efficient FPGA Acceleration of Convolutional Deep Neural Networks

Atul Rahman

This certifies that the thesis of Atul Rahman is approved.

06.14.2016

[illegible]

Thesis Supervisor: Jongeun Lee

[illegible]

Woongki Baek: Thesis Committee Member #1

[illegible]

Seokhyeong Kang: Thesis Committee Member #2

Abstract

Deep Convolutional Neural Networks (CNNs) are a powerful model for visual recognition tasks, but due to their very high computational requirement, acceleration is highly desired. FPGA accelerators for CNNs are typically built around one large MAC (multiply-accumulate) array, which is repeatedly used to perform the computation of all convolution layers, which can be quite diverse and complex. Thus a key challenge is how to design a common architecture that can perform well for all convolutional layers. In this paper we present a highly optimized and cost-effective 3D neuron array architecture that is a natural fit for convolutional layers, along with a parameter selection framework to optimize its parameters for any given CNN model. We show through theoretical as well as empirical analyses that structuring compute elements in a 3D rather than a 2D topology can lead to higher performance through an improved utilization of key FPGA resources. Our experimental results targeting a Virtex-7 FPGA demonstrate that our proposed technique can generate CNN accelerators that can outperform the state-of-the-art solution, by 1.80x to maximum 4.05x for 32-bit floating-point, and 16-bit fixed-point MAC implementation respectively for different CNN models. Additionally, our proposed technique can generate designs that are far more scalable in terms of compute resources. We also report on the energy consumption of our accelerator in comparison with a GPGPU implementation.

Contents

List of Figures

List of Tables

CHAPTER I

INTRODUCTION

Convolutional Neural Networks (CNNs) are one of the most common types of deep neural networks specialized for image recognition, such as object recognition [12] and semantic segmentation [13]. The superiority of CNNs for visual tasks is demonstrated by the dominance of CNNs among the top entries of recent ImageNet competition [17].

CNNs raise the amount of computation by orders of magnitude as compared with earlier multi-layer perception (MLP) models [9]. The increase in computation is caused mainly by the fact that (i) CNNs deal with raw image data directly, as opposed to using feature vectors as with MLPs, and (ii) CNNs typically have more layers than MLPs. The additional layers of CNNs consist of convolutional, pooling, activation, and normalization layers, and are found in the front side of a CNN, serving as a feature extractor. The output of the "feature extractor" is typically connected to a MLP-based classifier, which is the back-end of a CNN. Among all layers, convolutional layers account for the vast majority (about 90%) of computation [6]. Therefore how to accelerate convolutional layers is a critical problem.

Though GPGPUs (General-Purpose Gragphics Processing Units) are used more extensively for CNN acceleration today [5], FPGAs (Field-Programmable Gate Arrays) also have their advantage such as lower energy consumption. We focus on FPGAs in this paper but also provide energy estimation comparison against GPGPUs.

Previous work on FPGA acceleration of CNNs [1,2,8] shows progressively increasing throughput. FPGA based designs can either be computation or memory bandwidth bound. The work in [14] only considers optimizing FPGA based CNN design reducing the memory bandwidth requirement. However, for computation bound designs the proposed accelerator is not optimized. The parallelism scheme of [18] and [1] is same and the data reuse factor is very low due to not effective use of on-chip buffers. The approach in [8] was to accelerate CNN with both software and hardware implementation. The implementation of the processor was ad-hoc and not generalized design methodology for any given CNN topology and FPGA platform specific constraints. The most recent one [20] takes a high-level synthesis approach, and by optimizing for both computation rate and communication bandwidth, it can achieve 61.62 GFLOPS on a Virtex-7 FPGA when applied to a real-life CNN [12] with five convolutional layers, using 5 DSP units per multiply-add operation.

In this paper we present a highly optimized CNN accelerator architecture that can push the performance envelope even further, in terms of both computation and communication.

One key challenge in raising performance is how to design a common architecture that can perform well for *all* layers, overcoming the diversity and complexity of computations across layers (see Table 2.1). Fully parallel hardware implementation is simply impossible due to the sheer number of operations needed, making it crucial to choose the best parallelization scheme.

Our solution is a novel architecture called *Input-recycling Convolutional Array of Neurons (ICAN)*, which differs from previous solutions by parallelizing along *three carefully chosen dimensions* that mimics the convolution operation used in CNNs, leading to a 3D accelerator. While such a 3D accelerator could improve performance by better utilizing key FPGA resources, a potential drawback is its internal complexity. We address this problem with our *Input Reuse Network*, which can minimize routing complexity and exploit data reuse opportunity inherent in convolution operations. Another key factor contributing to the high efficiency of our solution is its aggressive parameter optimization considering *memory* as well as *compute* resources, which can significantly improve the computation-to-communication ratio of our accelerators.

We show through theoretical as well as empirical analyses that structuring compute elements in a 3D rather than a 2D topology can significantly improve the utilization of key FPGA resources such as DSP slices for various convolutional layer shapes. When applied to a real-life CNN [12] our ICAN accelerator can achieve in average 1.80x to over 4 times higher performance on a Virtex-7 FPGA compared with the state-of-the-art solution [20]. Moreover, when applied to a set of large CNNs, we find that the designs optimized by our algorithm are consistently better, with up to 35x improvement in one case, than those of the previous work. Finally, our energy estimation suggests that our FPGA approach can be more energy-efficient by about 11x to 13x than GPGPUs for convolutional layers.

The rest of the paper is organized as follows. In Chapter II, we explain and show theoretical analysis and architecture details. First, in Section 2.1, we consider the problem of maximizing the throughput of *any* accelerator that can be implemented on a modern FPGA, and in Section 2.2 propose our parallelization scheme along with performance analysis that is flexible yet highly efficient for convolutional layer computations. Section 2.3 presents the details of our accelerator architecture. The later Section 2.4 shows the process to find optimal architecture design parameters. In the following Chapter III , we present our experimental results, and discuss the related work in the next Chapter IV. Finally, we discuss about future work in Chapter V and conclude the paper in Chapter VI.

CHAPTER II

MAPPING CNNs on FPGA

2.1 Maximizing Computation Rate

2.1.1 DSP Utilization

Maximizing the computation rate on today's FPGAs often boils down to maximizing the utilization of DSP slices such as DSP48E1 in Xilinx Virtex-7 FPGA, as DSP slices are far more efficient than any other implementation on an FPGA for multiply-accumulate (MAC) operations. For instance, the mentioned DSP slice can perform a 25x18-bit multiplication followed by a 48-bit addition in just one cycle.

For any CNN accelerator implementing MAC operations using DSP slices only, DSP utilization can be calculated as:

$$DSP\ utilization = \frac{\#MAC\ units \times \#DSPs\ per\ MAC\ unit}{Total\ \#\ of\ available\ DSPs} \tag{II.1}$$

Here *#MAC units* is the number of MAC units in the CNN accelerator, which is a design parameter, and *#DSPs per MAC unit* is determined mainly by the precision of the MAC operation. Thus given an FPGA and the precision of a MAC operation, we can determine the maximum number of MAC units that can be included in an accelerator.

2.1.2 MAC Utilization

The other factor of the computation rate is *MAC utilization*, representing how many of the synthesized MAC units are actually used on average for useful computation. The key observation is that not every MAC unit in a design is utilized at all times, and that this utilization varies depending on how a CNN is mapped to the accelerator. Also note that whereas DSP utilization is a static value, which can be found in a synthesis report, MAC utilization varies from layer to layer. We compute the MAC utilization of a CNN from the total number cycles that it takes to complete the CNN, which is denoted by *#exec. cycles* in the following equation. For brevity we assume that MAC units have the throughput of one.

$$MAC\ utilization = \frac{Total\ \#\ of\ MAC\ ops\ in\ CNN}{\#MAC\ units \times \#exec.\ cycles} \tag{II.2}$$

Then the computation rate, which is proportional to the product of the two utilization factors, can determine the system throughput, provided that input and output are always ready. If not, any wait cycles for input/output need to be added to the total execution time.

While DSP utilization is considered in previous work as well, consideration of MAC utilization is new. We find that MAC utilization of the previous work [20] sometimes quite low, suggesting a scope for improvement. For further analysis of MAC utilization, we need to look into the structure of CNN computation.

2.1.3 Convolutional Layer Computation

Computationally a convolutional layer is a mere transformation of a 3D array into another 3D array using a series of 2D convolutions extended into the third dimension. Therefore computing one output point requires a 3-deep nested loop, and since the output is arranged in a 3D array we need a 6-deep nested loop to complete the computation of one convolutional layer as shown in Figure 2.1. The body of the loop nest is just one MAC operation as in matrix multiplication,

```
for ($m = 0;\ m < M;\ m{+}{+}$)
  for ($r = 0;\ r < R;\ r{+}{+}$)
    for ($c = 0;\ c < C;\ c{+}{+}$)
      for ($z = 0;\ z < Z;\ z{+}{+}$)
        for ($y = 0;\ y < K;\ y{+}{+}$)
          for ($x = 0;\ x < K;\ x{+}{+}$)
            $B[m][r][c] \mathrel{+}= W[m][z][y][x] \times A[z][Sr + y][Sc + x]$;
```

Figure 2.1: Computation of a convolutional layer.

Parameters	Description
Z	# of input feature maps
Y, X	Input height and width
M	# of output feature maps
R, C	Output height and width
K	Kernel size in one dimension
S	Stride in the input

Figure 2.2: Convoluational layer parameters

and all the loop levels are permutable if one handles the bias term or output initialization separately. The notations of convolutional layer parameters are shown in Figure 2.2.

The core of a CNN accelerator is an array of MAC units, which needs to be executed many cycles before completing one layer, which is why most CNN accelerators on an FPGA does one layer at a time. But since the same MAC array has to be used for all layers without FPGA reconfiguration, some of the MAC array will not be fully utilized, especially when the MAC array does the computation for the edge part of the original loop nest. This is the internal fragmentation issue in terms of MAC array utilization, and manifested as lower MAC utilization.

Thus a key question in mapping the loop nest is which loops to *parallelize* vs. which ones to *iterate*. *Parallelization* means replicating the hardware resources, thus requiring more DSP units, whereas *iteration* means the particular loop level is done sequentially, thus more execution cycles. Obviously parallelization is limited by the maximum number of MAC units derived from (II.1), but depending on exactly which loops are parallelized, MAC utilization will vary. Thus the goal of our accelerator design is to maximize MAC utilization for various layer shapes while not incurring high overhead in terms of implementation (e.g., routing) or on-chip buffer size requirement.

2.2 Our Proposed Acceleration Technique

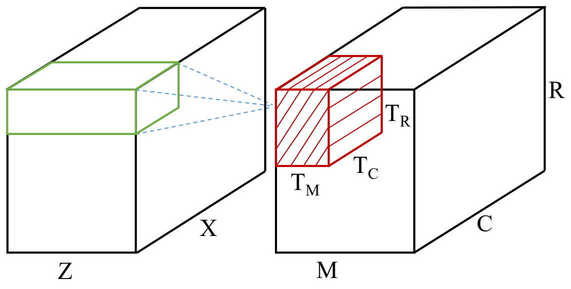
2.2.1 Our Parallelization Scheme

Figure 2.3a, which is a tiled and HW-unrolled version of a generic 6-deep convolutional layer code, illustrates our proposed parallelization scheme, including the selection of HW-unrolled loops (*which loops to parallelize*) and the order of loops (*in which order to iterate*). In the code, A, B, W are input, output, and weights, respectively.

Though the code might seem like a simple application of known loop transformations such as loop tiling, loop interchange, and loop unrolling, finding an optimal transformation can be

```
for (mm=0; mm<M; mm+=T_M)
 for (rr=0; rr <R; rr +=T_R)
  for (cc=0; cc<C; cc+=T_C)
   for (z=0 ; z <Z; z++)
    for (y=0; y<K; y++)
     for (x=0; x<K; x++)                HW UNROLL
      for (m=mm; m<min(mm+T_M,M); m++)
       for (r=rr; r<min(rr+T_R,R); r++)
        for (c=cc; c<min(cc+T_C,C); c++)
         B[m][r][c] += W[m][z][y][x] *
                         A[z][S*r+y][S*c +x];
```

(a) Our parallelization scheme in C code



(b) Our parallelization scheme

Figure 2.3: Our proposed parallelization scheme.

elusive due to the large number of combinations and the difficulty of quantitative evaluation with symbolic variables.

The hardware-unrolled iterations in Figure 2.3a are implemented as parallel compute elements, which are collectively referred to as *compute tile*. A compute tile is illustrated in red (shaded) in Figure 2.3b, where T_M, T_R, and T_C are the dimensions of the compute tile. The number of MAC units in our compute tile is $T_M T_R T_C$.

Our proposed architecture has the following unique features. First, our compute tile, by virtue of being 3D, has a higher degree of freedom to match the computation requirements of various convolutional layers, and has usually high MAC utilization as discussed in the next section. Second, the three dimensions of our compute tile match exactly the three dimensions of the output array, which is a natural and straightforward design. In other words, each MAC unit corresponds to one neuron given enough tile size, whereas in previous work, e.g., [20], one MAC unit corresponds to entire neurons in the RC plane of the output, creating artificial time-sharing of resources. Third, our compute tile boasts high reuse factors for input data (temporal reuse for m-loop, spatial reuse for r- and c-loops) and weight parameters (shared by all neurons in the RC plane). Finally, the on-chip buffer size requirement of our solution to realize maximal data reuse is not high in comparison to that of [20].

2.2.2 Performance Analysis

The number of MAC units of our proposed scheme is $T_M T_R T_C$. The number of computation cycles, T_{comp}, is:

$$T_{comp} = Z \cdot \left\lceil \frac{M}{T_M} \right\rceil \cdot K^2 \cdot \left\lceil \frac{R}{T_R} \right\rceil \cdot \left\lceil \frac{C}{T_C} \right\rceil \qquad \text{(II.3)}$$

Table 2.1: Convolutional layers of a CNN [12] & advantage of our scheme

	Layer parameters		#MAC	R/R_{th}	R/R_{th}
	(Z, Y, X)	(M, R, C)	Ops	(N=560)	(N=2800)
L1	$(3, 224, 224)$	$(48, 55, 55)$	105M	27.4	36.0
L2	$(48, 27, 27)$	$(128, 27, 27)$	224M	1.00	1.35
L3	$(256, 13, 13)$	$(192, 13, 13)$	150M	1/7.34	1/5.30
L4	$(192, 13, 13)$	$(192, 13, 13)$	112M	1/6.23	1/4.54
L5	$(192, 13, 13)$	$(128, 13, 13)$	75M	1/5.20	1/3.75

Now we can calculate MAC utilization,[1] or rather how much it deviates from the ideal, which stems from the ceiling operations in (II.3). We define deviation δ as $(T_{comp} - T_{ideal})/T_{ideal}$, where T_{ideal} is the ideal number of cycles, i.e., $T_{ideal} = Z \cdot \frac{M}{T_M} \cdot K^2 \cdot \frac{R}{T_R} \cdot \frac{C}{T_C}$. To facilitate calculation we introduce new variables $0 \le f_M, f_R, f_C < 1$, which are defined as $f_M = \left\lceil \frac{M}{T_M} \right\rceil - \frac{M}{T_M}$, with similar definitions for the others.

$$\begin{aligned}\delta &= \left(1 + \frac{T_M f_M}{M}\right)\left(1 + \frac{T_R f_R}{R}\right)\left(1 + \frac{T_C f_C}{C}\right) - 1 \\ &= \frac{T_M f_M}{M} + \frac{T_R f_R}{R} + \frac{T_C f_C}{C} + H.O.T. \\ &< \frac{T_M}{M} + \frac{T_R}{R} + \frac{T_C}{C} + H.O.T.\end{aligned}$$

In the same way we compute the deviation δ' for [20], which has a 2D tile consisting of $T'_M \times T'_Z$ MACs:

$$\delta' < \frac{T'_M}{M} + \frac{T'_Z}{Z} + H.O.T.$$

Assuming the same DSP utilization between the two designs, we have $T_M T_R T_C = T'_M T'_Z = N$. Often we have $R = C$, which however doesn't necessarily mean $T_R = T_C$. But for asymptotic analysis let us assume that $T_M = T_R = T_C = \sqrt[3]{N}$ and $T'_M = T'_Z = \sqrt{N}$. Then we can find the condition for *our* scheme's deviation to have a lower upper-bound than that of the previous work, i.e., $\delta < \delta'$:

$$\frac{T_M}{M} + \frac{T_R}{R} + \frac{T_C}{C} < \frac{T'_M}{M} + \frac{T'_Z}{Z} \qquad \text{(II.4)}$$

$$\sqrt[3]{N}\left(\frac{1}{M} + \frac{2}{R}\right) < \sqrt{N}\left(\frac{1}{M} + \frac{1}{Z}\right) \qquad \text{(II.5)}$$

[1]Here we assume that input/output arrays are ready, so the total number of execution cycles is solely determined by the computation cycles, T_{comp}.

$$\therefore\ R > 2\left(\frac{\sqrt[6]{N}-1}{M}+\frac{\sqrt[6]{N}}{Z}\right)^{-1} \triangleq R_{th} \tag{II.6}$$

$$R_{th} < 2\left(\frac{\sqrt[6]{N}-1}{M}+\frac{\sqrt[6]{N}-1}{Z}\right)^{-1} \tag{II.7}$$

$$\leq \frac{\sqrt{M\cdot Z}}{\sqrt[6]{N}-1} \tag{II.8}$$

The right side of (II.6), which is denoted by R_{th}, is bounded by (II.8) from the geometric-mean-harmonic-mean inequality. The inequality (II.8) suggests that the greater the output height (R) and width (C), or the higher the N, the more likely it is for our scheme to have higher MAC utilization. On the other hand, the greater the M and Z, the less likely for ours to be better, which agrees with our intuition, since M and Z are the dimensions taken by the previous work.

We compute the value of R_{th} in (II.6) for a real CNN, AlexNet [12]. We use two values of N (i.e., 560 and 2800), relevant for our target FPGA. The two rightmost columns of Table 2.1 show the ratio, R/R_{th}. The higher it is, the better our scheme will be compared with the previous work. The table predicts that our scheme will be better in layers 1 and 2, which account for nearly 50% of the computation in convolutional layers. In reality, the actual MAC utilization is determined by the selection of the T parameters, and since our scheme has simply more possible combinations to try, our scheme tends to be better than what these numbers may suggest.

2.3 Architecture Details

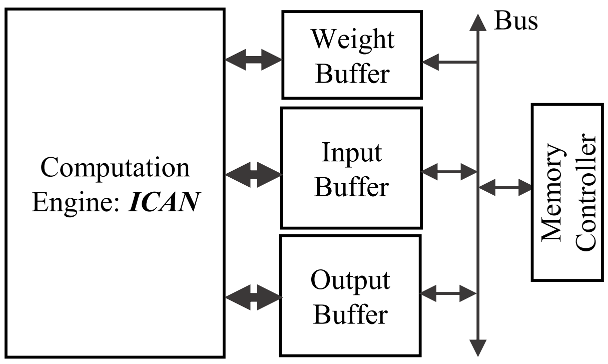


Figure 2.4: The top-level view of our accelerator architecture, which does one layer computation at a time. All three buffers are double-buffered.

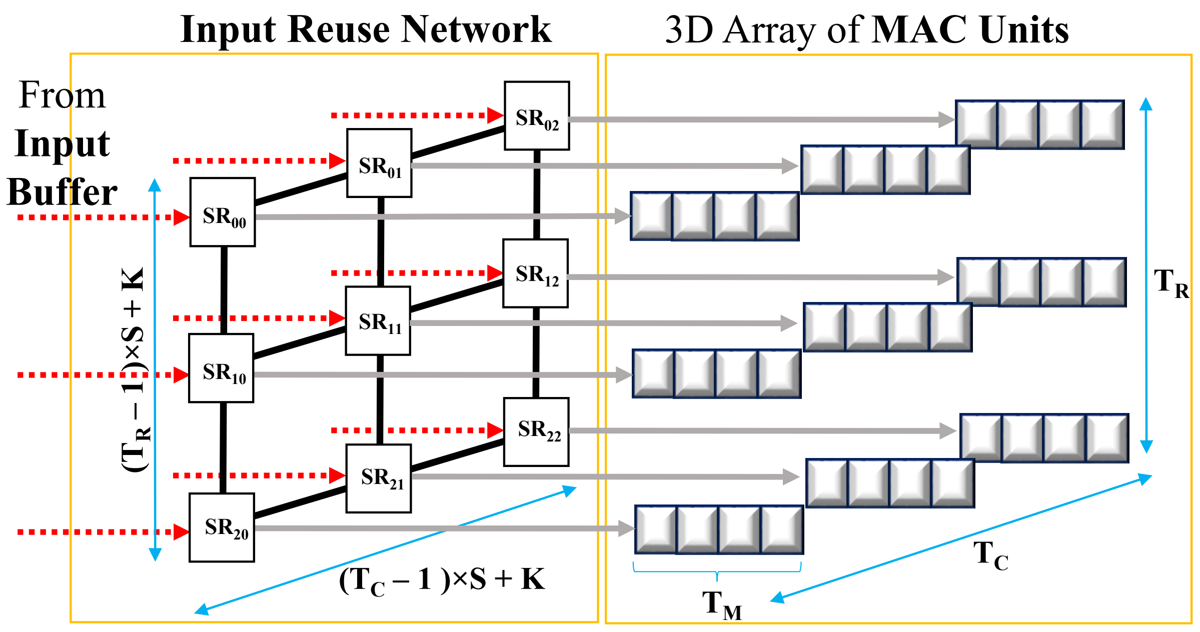


Figure 2.5: Main components of ICAN: input reuse network and compute tile. Compute tile, which is a 3D array of $T_M T_R T_C$ MAC units, takes additional input from the weight buffer, and its output is stored in the output buffer.

2.3.1 Architecture Overview

Figure 2.4 illustrates our DNN accelerator architecture. It implements one convolutional layer computation without including max-pooling or activation functions. The same datapath is used for every convolutional layer of a DNN, and the specifics of each layer such as different input/output/kernel sizes are taken care of by hardware controller. All data are stored in the external DRAM, including input/output feature maps and weight parameters. For faster processing, the computation engine, *ICAN*, uses 3 on-chip buffers, *input*, *output*, and *weight*, which are all double-buffered to hide external memory access latency. In addition to what is shown, there is a small processor for miscellaneous tasks such as host interface and memory initialization.

2.3.2 ICAN: Input-recycling Convolutional Array of Neurons

Being a 3D-array, the internal architecture and routing of a computation engine could be quite complicated. Additional challenge is how to exploit data reuse, in particular the spatial reuse along the r- and c-loops of Figure 2.3a.

Figure 2.5 illustrates our proposed ICAN architecture, consisting of an *input reuse network* and a *compute tile*. In addition, *shape adapter* is needed to interface with the input buffer. Our compute tile is a 3D array of $T_M T_R T_C$ MAC units with no connection among the elements. The *input reuse network* is a set of registers connected in a 2D-torus[1] interconnect. One register is connected to T_M MAC units of the compute tile, which works in a SIMD fashion. The input reuse network can be loaded very quickly from the input buffer, and provide input data for the connected MAC units during the next K^2 cycles. This can be done by making each value traverse its (K, K) neighbors—for instance, by shifting the register values horizontally first (say, to the west) for the first $(K - 1)$ cycles, and then vertically for one cycle, and then reverse-horizontally (to the east) for the $(K - 1)$ cycles, and so on. Note that the entire 2D array is shifted simultaneously as in a systolic array, which simplifies control.

To correctly implement the computation of HW-unrolled iterations in Figure 2.3a, we need $(K-1)$ extra registers along the eastern edge and the southern edge of the input reuse network, which are called *guard registers*. Therefore the total number of registers are $T_R T_C + K(T_R + T_C)$ if stride is 1, or $((T_R - 1)S + 1)\,((T_C - 1)S + 1) + K\,((T_R - 1)S + 1 + (T_C - 1)S + 1)$ for the stride of S. If the stride is greater than 1, each MAC unit is still connected to just one register[2] and the data in the input reuse network are shifted in the exactly same way. The reuse factor will decrease however, but it is due to our parallelization scheme, not our ICAN design.

A *shape adapter* is needed between input buffer and input reuse network to match between the elements of the two 2D arrays. It is simply a 2D array of isolated registers with muxes for handling zero-padding in boundary tiles. The control inputs for the muxes cannot be shared in general, increasing the complexity and area overhead of the *Read Controller*[3]. On the other hand, the shape adapter is needed only once in K^2 cycles, making its latency easily hidden.

Without the input reuse network, a naïve implementation would require K^2 connections for each subarray of T_M MAC units, requring K^2 times more wiring than ours.

The MAC units accumulate all the product terms until the next time-multiplex point (i.e., at every iteration of the *cc*-loop), at which point all the results are written back to the output buffer in a few cycles, followed by loading the next set of output values from the output buffer (our implementation allows 1-cycle simultaneous load/store).

2.3.3 Data Tiles and On-chip Buffer Design

When a design is IO-bound, the amount of external data transfer can have a significant impact on the achievable throughput. Though convolution layers are usually compute-bound especially when we look at them in isolation, we must design one accelerator for all layers of a

[1]The wrap-around is needed either vertically or horizontally, but not both.

[2]In practice, multiplexers are needed if a DNN has layers with different strides because one accelerator must support all convolutional layers of a DNN.

[3]We find that our Read Controller for designs of Section 3.2 uses about 9% of the LUT.

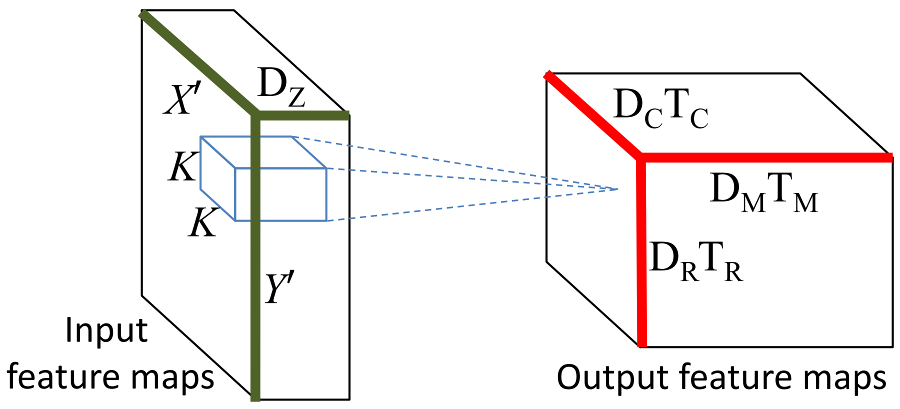


Figure 2.6: Output data tile (red) measuring $D_M T_M \times D_R T_R \times D_C T_C$ pixels, and input data tile (green) having D_Z pixels in the input feature map direction.

given network, which can cause some layers to become IO-bound. To minimize the amount of external data transfer, using large on-chip buffers can help.

We introduce *data tile* that is the block of data kept on-chip. We define three data tiles: input, output, and weight tiles. Data tile size is defined as some multiple of compute tile size. The multiplicative factors are given by these four parameters, D_M, D_R, D_C, and D_Z (see Figure 2.6). The *output tile* is simply (D_M, D_R, D_C) times the output of one compute tile, and its size is $B_{out} = D_M T_M \cdot D_R T_R \cdot D_C T_C$ words. The *input tile* is defined as the portion of input data needed to compute the output tile, and its size is $B_{in} = D_Z Y' X'$ where $Y' = ((D_R T_R - 1)S + K)$ and $X' = ((D_C T_C - 1)S + K)$. The *weight tile* is defined similarly, and its size is $B_w = D_Z D_M T_M K^2$.

On-chip buffers are the physical realization of these data tiles, with double buffering to hide the external memory access latency. The size of on-chip buffers can be greater than that of data tiles, due to the constraint that the depth of a buffer must be a power of two. Also there is a limit to the width of an SRAM block that can be synthesized on an FPGA, which is denoted by *MaxMemWidth*. Thus it is advantageous to use the smallest width and the largest depth for a given size.

The widths (in words) of on-chip buffers are set in terms of T-parameters as follows. This is so that the compute array can read and write a certain number of words simultaneously.

$$B_{out}^{width} = T_M T_R T_C \tag{II.9}$$

$$B_{w}^{width} = T_M \tag{II.10}$$

$$B_{in}^{width} = T_R T_C \tag{II.11}$$

If the width is larger than *MaxMemWidth*, we implement the width by synthesizing multiple SRAMs, which are used together like a multi-bank memory. All the banks comprising one buffer

```
for $(m_2 = 0;\ m_2 < M;\ m_2 \mathrel{+}= D_M T_M)$
for $(r_2 = 0;\ r_2 < R;\ r_2 \mathrel{+}= D_R T_R)$
for $(c_2 = 0;\ c_2 < C;\ c_2 \mathrel{+}= D_C T_C)$
for $(z_2 = 0;\ z_2 < Z;\ z_2 \mathrel{+}= D_Z)$

for $(m_1 = m_2;\ m_1 < \min(M, m_2 + D_M T_M);\ m_1 \mathrel{+}= T_M)$
for $(r_1 = r_2;\ r_1 < \min(R, r_2 + D_R T_R);\ r_1 \mathrel{+}= T_R)$
for $(c_1 = c_2;\ c_1 < \min(C, c_2 + D_C T_C);\ c_1 \mathrel{+}= T_C)$
for $(z_1 = z_2;\ z_1 < \min(Z, z_2 + D_Z);\ z_1{+}{+})$
for $(y = 0;\ y < K;\ y{+}{+})$
for $(x = 0;\ x < K;\ x{+}{+})$

for $(m = m_1;\ m < \min(M, m_1 + T_M);\ m{+}{+})$ // unroll
for $(r = r_1;\ r < \min(R, r_1 + T_R);\ r{+}{+})$ // unroll
for $(c = c_1;\ c < \min(C, c_1 + T_C);\ c{+}{+})$ // unroll
    $B[m][r][c] \mathrel{+}= W[m][z_1][y][x] \times A[z_1][Sr + y][Sc + x];$
```

Figure 2.7: Convolutional layer computation, after data tiling

has the same depth, so the choice of widths for the banks makes little difference in terms of SRAM usage.

Now the depth of a buffer can be found from the buffer width and the size of the data tile. There are two considerations. First, the depth must be a power of two. Second, sometimes a layer dimension (e.g., M) may be less than the product of the corresponding T- and D-parameters (e.g., $D_M T_M$). In this case the needed buffer size follows the layer dimension, not the product of T- and D-parameters. This may happen in some layers only, but when it does, it can affect the required buffer size quite dramatically. So it is important to make the distinction.

Let us define *effective* data tile parameters as follows: $d_M = \min(\lceil M/T_M \rceil, D_M)$. Other parameters, d_R, d_C, and d_Z, are defined similarly. Then the depths of the buffers are given as:

$$\begin{aligned}
B_{out}^{depth} &\geq d_M \cdot d_R \cdot d_C \\
B_{w}^{depth} &\geq d_M \cdot D_Z \cdot K^2 \\
B_{in}^{depth} &\geq D_Z \left\lceil \frac{S(d_R T_R - 1) + K}{T_R T_C} \right\rceil (S(d_C T_C - 1) + K)\,.
\end{aligned}$$

The depth parameter is determined to be the smallest power of two, satisfying the above inequality for all layers. The buffer size is the product of its width and its depth (times two due to double buffering). The sum of all buffer sizes must not be greater than the available on-chip memory size.[1]

[1]Exact estimation of on-chip memory usage on FPGAs is extremely difficult due to the complex interplay between memory latency, parallelism, bit-width, memory partitions, etc. Thus we set the available on-chip memory size conservatively.

We show the generic convolutional layer computation code including the data tiling in Figure 2.7. The trip count τ of each buffer, or how many times they have to be refilled, is given as follows:

$$\begin{aligned}
\tau_{out} &= \left\lceil \frac{M}{D_M T_M} \right\rceil \left\lceil \frac{R}{D_R T_R} \right\rceil \left\lceil \frac{C}{D_C T_C} \right\rceil \\
\tau_{in} &= \tau_{out} \left\lceil \frac{Z}{D_Z} \right\rceil \\
\tau_w &= \tau_{in}.
\end{aligned}$$

These can be used to calculate the amount of external memory access, from which the data transfer time, T_{dt}, can be determined based on the empirical affine relationship between memory access latency and total data transfer size. The data transfer size, $DataTransSize$ of each buffer is the product of buffer size and trip. The total data transfer size, $DataTransSize^l$ of a layer l is simply the summation of the data transfer size of all the buffers.

2.4 Finding Optimal Parameters

We have constraints on DSP, on-chip memory usage, and available bandwidth of the external memory. The DSP usage constraint is that $T_M T_R T_C \leq \#MAC\ units$, where $\#MAC\ units$ comes from (II.1). On-chip memory (i.e., BRAM) usage constraint is that the combined size of the three data tiles must be within half the available BRAM size (half because of double buffering). The available bandwidth constraint is BW_limit under operating frequency f_{CLK}.

Finding the best parameter combination can be formulated as an optimization problem. The decision variables are 3 compute tile parameters (T_M, T_R, T_C) and 4 data tile parameters (D_M, D_R, D_C, D_Z). Frist, the objective is to minimize the total execution cycles (T), which is the sum of the execution cycles of each layer (l) i.e. $T = \sum_l T^l$. The number of execution cycles of a layer is determined as follows:

$$T^l = \max(T^l_{comp}, T^l_{dt}) \tag{II.12}$$

$$\text{where as, } T^l_{dt} = DataTransSize^l \cdot f_{CLK} / BW\_limit \tag{II.13}$$

Secondly, the objective is to minimize the required bandwidth of the design which is shown in (II.14).

$$BW = \max_l DataTransSize^l / T^l \tag{II.14}$$

And, thridly, the objective is to minimize the on-chip memory usage.

Taking the maximum, though hardly new, is a noticeable departure from the prior state of the art work [20], and can make a difference in the solution quality if the network/layer is IO-bound. Previously if a combination has a higher T_{dt} than T_{comp}, it was simply discarded; in other words, there was a hidden constraint that says $T_{dt} \leq T_{comp}$, making it suboptimal. Our new formulation is based on small hardware that stalls the compute array if the data transfer has not completed, but this hardware would be needed anyway to ensure proper operation of double buffering.

We solve the problem using an exhaustive search. The compute tile parameters are upper-bounded by their respective layer parameters, e.g., $T_M \leq M$. Products of compute/data tile parameters are similarly bounded, e.g., $D_M T_M \leq M$ (see Figure 2.7). Our implementation of exhaustive search-based exploration for AlexNet [12] takes less than ten minutes on a modern workstation using a single-thread execution for each of the designs in Section 3.2.

CHAPTER III

Experiments

We now present our synthesis result of the ICAN architecture for 32-bit fixed-point precision, followed by performance results for different precisions and for different CNN graphs. Finally we present energy comparison results with GPGPU. We end this chapter with further comparison with LUT-DSP based MAC approach of a recent research vs our DSP based MAC implementation.

3.1 Synthesis Result

We have implemented our CNN accelerator in Verilog and synthesized it for Virtex-7 XC7VX485T-2 on a VC707 FPGA board. Vivado 2015.2 is used for simulation and synthesis. The output buffer is implemented with *simple* dual-port BRAMs, which have very small overhead over the single port version. For the external memory controller bus we use the AXI Multi-Ported Memory Controller (MPMC) generated by MIG IP block of the tool.

Table 3.1 shows the synthesis result for the ICAN architecture implemented using 32-bit fixed-point MAC units each containing 5 DSP slices. We use the optimal architecture parameters

Table 3.1: Synthesis result of ICAN with $(T_M,T_R,T_C) = (11,7,7)$

Resource type	LUT	FF	DSP	BRAM
Utilization	9.22%	7.28%	96.25%	46.89%

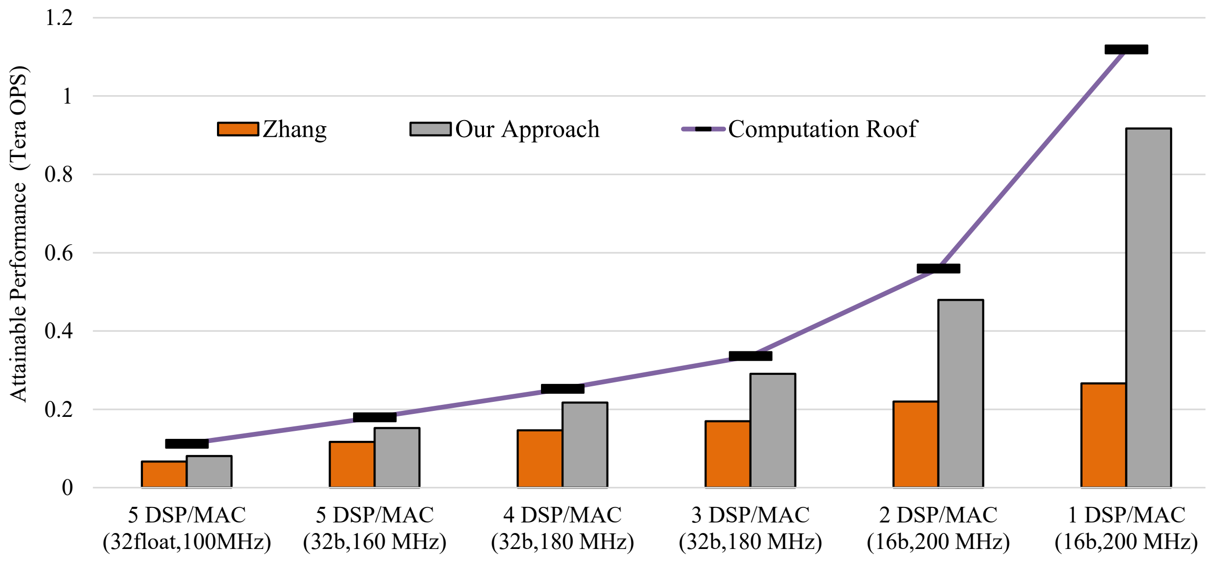


Figure 3.1: Attainable performance for different MAC options.

supporting all the convolution layers of the Alexnet [12] under DSP resource constraint set at 2,700 slices. This design can achieve 147.82 GOPS performance throughput. The critical path of the design are the MAC units, for which the operating frequency of 160MHz is achieved. At this frequency our Vivado simulation with an MPMC module and a 64-bit DDR3 shows the external memory bandwidth of 6.2GB/s. The LUT (look-up-table) consumption by ICAN containing 539 MAC units is less than 10%, most of which is due to input reuse network. This synthesis report does not include the shape adapter as most of the shape adapter is implemented in the read controller. For the above design point, we have also synthesized on-chip buffers separately, whose optimal size is found to be (D_Z, D_M, D_R, D_C) = (16, 3, 2, 2). The BRAM usage (including double buffering) is less than 50%.

3.2 Compute Scalability: Exploring Different MAC Options

We evaluate a number of different MAC options to see the scalability of our architecture for higher compute resources. By reducing the precision we can design a MAC using 1 to 5 DSP slices, enabling up to 5x more MAC units. This is motivated by the fact that in practice 16-bit fixed-point implementation gives enough precision in terms of recognition accuracy for many convolutional neural networks including AlexNet. These five options are all fixed-point MACs of either 32-bit or 16-bit, with different clock frequency and maximum memory bandwidth. In addition we consider a 32-bit floating-point MAC option, running at mere 100MHz using 5 DSPs/MAC. For the floating-point case only, we limit the DSP utilization to 80% for easier

Table 3.2: DSP utilization comparison for Figure 3.1

#DSPs/MAC	5-float	5-fixed	4	3	2	1
Previous [20]	80.00	91.43	91.43	96.00	99.43	82.89*
Proposed	80.00	100.00	96.00	96.00	98.00	98.00

*The previous work [20] uses adder trees to do summation along the Z direction. About half the adders cannot be combined with multipliers *in the one DSP-per-MAC case*; we assume LUTs are used to implement those adders.

comparison with the previous work [20].

Figure 3.1 shows the performance comparison result. For the floating-point case we find that our MAC utilization is 1.22x (22% higher) compared with that of the previous work, resulting in the performance of 80.78 GFLOPS vs. 66.3 GFLOPS.[1] Furthermore the attainable performance gap between ours and the previous work increases as the precision reduces, or as the number of available MACs increases, up to 3.41x for 1 DSP-per-MAC case.

The graph also shows the computation roof, which is the maximum achievable performance for unlimited memory bandwidth. For our designs, the performance lies close to the computation roof.

In all the cases except for 1 DSP-per-MAC, both designs are computation-bound. In other words, the difference due to our better on-chip memory management does not play a role except for the 1 DSP-per-MAC case; the performance is determined simply by the product of MAC utilization and DSP utilization (with weighting factors due to differences in the amount of computation among layers). Table 3.2 compares the DSP utilization of the two schemes, which also implies that the average MAC utilization (=Performance ratio / DSP utilization) is quite higher for our scheme, actually much higher than the rough criterion of (II.8) suggests. In the 1 DSP-per-MAC case, the DSP utilization could go higher if not for the memory bandwidth limit.

Another key factor for this performance difference is that the MAC utilization of the previous work is substantially lower in layer 1 (our architecture has almost the same MAC utilization across all layers). This is because for the first layer, $(Z, M) = (3, 48)$, which means that in their architecture the change in number of MAC units beyond (3×48) doesn't help increase the performance. This may be the peculiarity of the first layer, but most CNNs necessarily have very small numbers of input/output feature maps and very large input/output sizes (i.e., horizontally/vertically) in layer 1. In fact, for later layers (e.g., layers 4 and 5) we do observe that our architecture has lower performance than the previous work in the 5 DSP-per-MAC case.

[1]The number 66.3 GFLOPS is based on our estimation, and higher than what is reported in the paper (=61.62) but matches with the number from other sources.

Table 3.3: CNN models (The 2nd column shows the number of convolutional layers)

CNN	#Conv. Layers (#MAC Ops)	Range of (Z,M,R,C)
AlexNet [12]	5 (0.67 B)	(3, 48, 13, 13) – (256, 192, 55, 55)
SpeedSign [14]	3 (2.70 B)	(1, 6, 173, 313) – (16, 80, 715, 1275)
StreetScene [7]	4 (6.55 B)	(3, 12, 103, 71) – (48, 128, 492, 367)
VGG-A [19]	8 (7.49 B)	(3, 64, 14, 14) – (512, 512, 224, 224)
VGG-E [19]	16 (19.50 B)	(3, 64, 14, 14) – (512, 512, 224, 224)

Table 3.4: Performance comparison (in GOPS) for different CNN models

	16-bit fixed-point			32-bit fixed-point			32-bit floating-point		
CNN	[20]	Our	Ratio	[20]	Our	Ratio	[20]	Our	Ratio
AlexNet	269.71	918.58	3.41	146.32	217.37	1.49	66.38	80.87	1.22
SpeedSign	25.60	896.16	35.01	22.32	226.48	10.15	12.03	85.48	7.11
StreetScene	127.94	1000.82	7.82	90.13	230.89	2.56	41.29	85.20	2.06
VGG-A	914.54	1020.87	1.12	229.51	239.06	1.04	86.11	89.04	1.03
VGG-E	987.43	1034.80	1.05	232.12	239.70	1.03	86.73	89.38	1.03
Geometric mean	240.08	973.32	4.05	109.421	230.55	2.11	47.67	85.94	1.80
Ideal performance		1120.00			252.00			89.60	

3.3 Results for Different CNNs

We evaluate our architecture for different CNNs as listed in Table 3.3. These, including AlexNet, are all real-life, large CNNs, with 5 to 16 convolution layers each. We use all the convolution layers. In addition to the different numbers of layers, these CNNs have different ranges of values in their Z, M, R, and C parameters, making an interesting comparison. To make it more interesting we consider three precision levels including one floating-point case (5 DSPs/MAC, 100MHz, 80% DSP utilization budget) and two fixed-point cases (1 and 4 DSPs/MAC with 200MHz and 180MHz, respectively; 100% DSP utilization for either).

Table 3.4 compares the designs generated by the previous work [20] vs. our proposed method. The performance in GOPS is obtained from the total number of cycles for all the convolution layers of a network, and thus representative of real performance. In all cases the table demonstrate that our method can generate designs that are consistently better than the previous work, and close to the ideal performance, which is computed without the ceiling operators in the performance model. The performance ratio is often significant, up to 35x for SpeedSign.

Across different precision levels, the relative advantage of our method over the previous work increases as the number of DSPs per MAC decreases, or equivalently as the number of MAC units available increases. This is because being 3D, our designs can make much better uses of MAC units than the previous work. As a result, with our method the MAC utilization always remains high, at least 80% for every case, whereas that of the previous work can go as low as

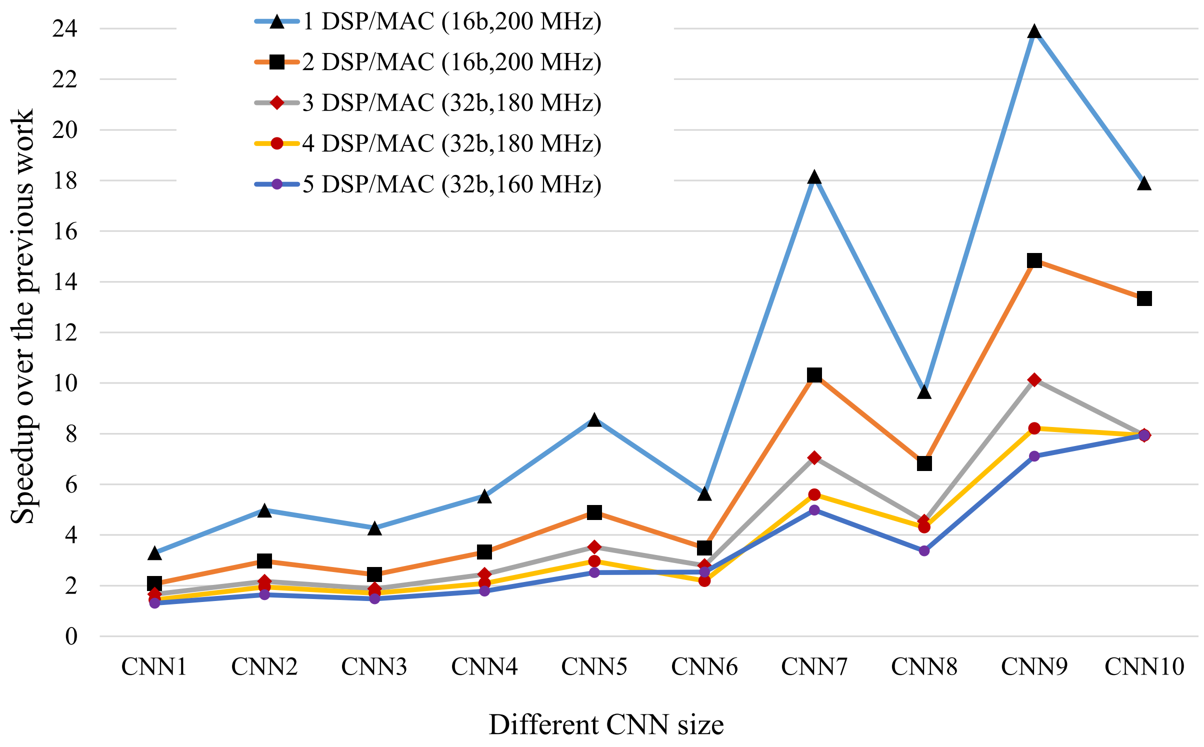


Figure 3.2: Performance ratio of ours over the previous work [20] for different CNN graphs.

5% (aggregated for all layers) in the case of SpeedSign at 16-bit precision.

The Z-M parallelization employed by the previous work proves to be a particularly unlucky choice for SpeedSign. First, SpeedSign has rather low values of $Z \leq 16$ and $M \leq 80$. So even the best—which in this case means the greatest—compute-tile parameters $(T_Z, T_M) = (16, 80)$ can utilize only 45% of the available DSP slices in the 16-bit case. Second, the actual Z and M values in the front layers are even smaller, with $Z = 1$ in the first layer. This results in many DSPs being actually idle for many cycles, dropping the overall MAC utilization to 5%. Combined, the previous method is able to achieve only 2.3% of the ideal performance, explaining the 35x difference in throughput.

The small improvement by our method for VGG-A and VGG-E is because the previous method is already close to optimal, which in turn is because Z and M parameters are consistently higher across layers for those models except for the first layer (see Table 3.3).

As evident by now, the CNN topology affects the tile size selection. We further show that our approach is far more efficient for smaller CNN graphs as well. We evaluate our proposed general methodology for different *small* synthetic CNN topologies (scaled down from AlexNet) and compare with the previous work for different type of fixed-point MAC units. We scale down the AlexNet to 10 different CNN topologies. The number of output feature maps, the

Table 3.5: Estimated energy consumption of GPGPU vs. FPGA

Type	Runtime	Power	Energy
Titan X (dissipating 60% of TDP)	21.26ms	150W	3.19J
Titan X (dissipating 50% of TDP)	21.26ms	125W	2.66J
Virtex7 FPGA + DDR3	16.46ms	14.3W	0.235J

size of input image, and stride are changed. For each scale-down, we reduce the image size and number of output feature maps by 10% while keeping the input image depth fixed to 3 (i.e., RGB image). In AlexNet, the stride for all layers is 1 except for the first layer, which we do not change. We change the *stride* of the first layer sequentially as we scale down the net. The sequence of stride is chosen as (4,4,3,3,3,2,2,1,1,1) for 10 different CNN topologies from the largest (*CNN1*) to the smallest size (*CNN10*). We keep the same kernel matrix size, and the input feature maps and row/column of input/output feature maps are determined according to the rule of the convolution operation, stride, and max-pooling architecture of the original DNN.

For each of the 10 CNN topologies we show our performance gain in terms of attainable performance ratio in Figure 3.2. This comparison shows that our proposed architecture template which uses FPGA resources efficiently is even scalable across small CNN topologies unlike that of [20]. This is because we can reach the minimum data transfer time in case of IO-bound designs by utilizing all the available on-chip memory while in the previous work there is always ($M \times T_M$) memory access overhead for input buffer. As a consequence, the attainable throughput of the previous approach is limited by the memory bandwidth, which in turn reduces the DSP utilization—to less than 40% for the two smallest graphs—whereas the DSP utilization for our architecture remains higher than 90% for all sizes.

3.4 Energy Comparison with GPGPU

We compare the energy efficiency of the proposed solution against that of the GPGPU. Titan X is one of the most preferred GPUs today for machine learning, and optimized for single-precision floating-point operations (double-precision is not supported). For FPGA we use the VC707 platform with 32-bit floating-point design, which is identical to the floating-point case in the previous section. For fair comparison we consider convolutional layers only, for both cases.

For GPGPU we use Caffe [11], a flexible yet highly optimized deep learning framework. We measure the runtime of 5 convolutional layers of AlexNet during 30 inference runs using three different images. The first two runs for each image may sometimes record unusually high runtime; those aberrations are excluded when taking average. From the average runtime we calculate energy, assuming the actual power dissipation is 50%   60% TDP (Thermal Design Power). The TDP of Titan X is 250W.

We estimate FPGA power dissipation with Xilinx Power Estimator (XPE) [10] using synthesis report (i.e., resource usage) and 60% toggle rate everywhere except for clock toggle rate which is 100%. The average power due to FPGA itself is estimated to be 13.65W. We also consider the external DRAM, a 1GB DDR3, whose power is estimated with DRAMPower [3] using read/write traces generated by our bit-true accelerator simulator.

Table 3.5 shows the results. Surprisingly our FPGA design, which runs at mere 100MHz, wins even the runtime comparison. And because the FPGA is 9x to 10x more power-efficient than the GPGPU, our FPGA design is found to be 11.3 to 13.6 times more energy-efficient than the GPGPU option. Part of this impressive difference is due to the fact that we consider convolution layers only. Fully-connected layers may require more external memory access, reducing energy efficiency of our solution. The other layers such as max-pooling and normalization may also dilute the FPGA advantage. Also since inference takes very little time compared with training, it may have received little optimization effort during Caffe development.

On the other hand, we allow only 80% of DSP slices in our experiment. With the reserved 20% we can either implement other layers or further improve the energy efficiency. Also our FPGA approach allows for fixed-point designs, which can further drive up energy efficiency dramatically. These findings demonstrate the superiority of our FPGA designs compared with GPGPU, and reinforce the need for research into FPGA-based accelerators for deep neural networks.

3.5 Comparison with LUT-based MAC Approach

Recently a new CNN accelerator [16] is proposed that uses LUTs in addition to DSP blocks to implement MAC units. It also features end-to-end implementation of a CNN model, a modified version of VGG-D [19], with 16 parameter layers including 13 convolution layers. Here we provide a quantitative comparison based on their published results and our estimated performance.

Table 3.6: Performance comparison with [16] (LUT-based MAC units are used in [16] but not in ours)

	#Ops ($\times 10^9$)	Theoretical GOPS reported in [16]	Our GOPS estimated
Convolution layers	30.69	249.31	266.53
Fully connected layers	0.073	2.47	2.15
Overall	30.76	201.51	206.30

The FPGA platform is a Zynq board with Kintex running at 150 MHz and 4.2 GB/s memory bandwidth. We use the same setting. For precision 16-bit fixed-point is used. For non-convolution layers we assume that max-pooling, ReLU, and LRN (Local Response Normalization) layers can be mapped using LUTs without using DSP blocks. Fully connected layers are assumed to run on our accelerator utilizing only a fraction of the MAC units due to the memory bandwidth limit.[1]

Table 3.6 compares performance, which is without considering LUT-based MAC for our architecture. Allowing LUTs for implementing MACs can further increase our performance. There are a few factors contributing to the surprisingly high performance of our architecture. First, our architecture parameters such as T-parameters and D-parameters are optimized through an exhaustive search, whereas such an optimal parameter selection would be hard in the case of [16]. Another reason why ours can perform better may be that our architecture is optimized for convolution layers only, whereas that of [16] is optimized for all layers.

[1]The bandwidth 4.2 GB/s translates into 22 words/cycle where one word equals 16 bits. We conservatively assume that we need to read 2 words and write 1 word per every MAC operation, which allows us to do 7 MAC operations per cycle.

CHAPTER IV

Related Work

The bottleneck in maximizing performance of a convolutional layer of a FPGA based CNN accelerator is due to two reasons, 1) The compute time is higher than the data transfer time i.e. compute bound or 2) the data transfer time is higher than the compute time i.e. memory bound. One or more convolutional layers of a CNN can be either compute or memory bound making it important to optimize the accelerator hardware taking into account both bottlenecks to maximize the total performance of all the convolutional layers of a CNN.

The approach in [8] is to accelerate CNN with both software and hardware implementation focusing on optimization for compute bound case. However, the compute bound optimization is inferior, as the 2D convolution unit is realized with 2D array structure that depends on the kernel size which clearly limits the efficient use of computation resources given that the kernel size in practical CNNs is very small in compared to the maximum number of available DSP slices of modern FPGA chip. The implementation of the processor is ad-hoc and not generalized design methodology for any given CNN topology and FPGA platform specific constraints. The work doesn't consider efficient on-chip memory utilization in case of memory bandwidth bound design. Likewise, [18], [1], and [2] optimizes only for compute bound case that implements complete CNN graph on FPGA with different parallelization schemes.

The work in [14] optimizes the accelerator architecture to reduce the data transfer time. However, the proposed architecture design ignores optimizing architecture for compute bound designs. Three different parallelism schemes and architectures are proposed out of which one architecture is selected and reconfigured for each convolutional layer. The reconfiguration time after each convolutional layer affects the performance due to increased latency. They propose to effectively reuse the data and utilize on-chip memory. However, they use soft-processor to load and store data to/from external memory which obstructs efficient use of maximum available bandwidth of external memory. A more general architecture and effective on-chip memory utilization scheme is proposed in the work of [15]. They provide methodology for loop transformations such as loop tiling and interchange focusing on inter tile data reuse. Though the methodology is generic for any application, it is not optimized specially for the convolutional layer computation. It modifies the architecture of [14] for convolutional layer computation and further improves the utilization of on-chip memory and hence the required bandwidth. However, the parallelizing scheme of the compute engine or in other words the loop unrolling method is unchanged and so is also not optimized for the most frequent case that is compute bound convolutional layers.

The work in [20] provides a more holistic design methodology tailored for convolutional layers and that outperforms the prior works. By carefully designing the compute tile to maximize data reuse, and by considering both the external memory bandwidth and the amount of compute resources on the FPGA platform, it can achieve performance exceeding that of any other previous work. Another advantage of [20] is that it is an HLS (High Level Synthesis)-based approach, which means that the RTL design can be obtained through an HLS tool. However their proposed compute tile is 2D, which has limited MAC utilization as demonstrated in our experiments. Furthermore they do not try to utilize on-chip memories maximally, and therefore may not be able to achieve the lowest memory bandwidth requirement, which can be critical for memory-bound designs.Our proposed three compute tile T-parameters, and four D-parameters vs two T-parameters, and two D-parameters of [20] in average minimizes both the compute and data transfer time for different arbitrary real life CNN graphs.

Hardware accelerators for CNN have been implemented in ASICs as well [4]. Though ASIC implementations have advantages over FPGA such as higher compute density, higher energy efficiency, and higher external memory bandwidth, the lack of reprogrammability is a big downside.

CHAPTER V

Future Work

First, one of the limitation of this work is the CNN accelerator design time as the design process with Hardware Description Language (HDL) takes long time and thus FPGA based accelerator remains less user friendly than the counterpart GPGPU. The future objective will be directed towards building an end-to-end, easy to use, and generic FPGA based accelerator design framework that would emulate our proposed CNN hardware accelerator design technique. The recent inception of High Level Synthesis tool reduces the hardware design time and hence, it is one of the viable approach for the framework.

Secondly, the on-chip buffer size calculation technique needs to be improved and more accurate to better constraint the maximum on-chip buffer size specially for the IO-bound CNN accelerator design.

And lastly, the convolution in frequency domain can benefit the acceleration of convolutional layer computation due to the decrement of the number of operations. The recent state-of-the art CNN models are good candidate for the frequency domain convolution and that too for FPGA based CNN accelerators.

CHAPTER VI

Conclusion

In this paper we presented *ICAN*, our novel accelerator architecture tailored for convolutional neural networks. ICAN is a 3D compute tile that is highly efficient yet flexible for a range of convolutional layer shapes. Being a 3D compute tile, it is a more natural fit for the convolutional layer computation, where the input and output are given in 3D arrays. To address the challenge of complex internal wiring and complex input reuse patterns, we propose an Input Reuse Network that is a simple 2D mesh-like array of registers. Our evaluation mapping the convolutional layers of real-life CNNs demonstrate that our accelerator can achive in average from approximately 1.80x to maximum 4.05x performance for 32-bit floating-point, and 16-bit fixed-point MAC case respectively on a Virtex-7 FPGA compared with the previous work. Additionally, our designs are far more scalable in terms of compute resources and CNN size.

References

[1] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. A programmable parallel accelerator for learning and classification. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 273–284, New York, NY, USA, 2010. ACM. 1, 2, 24

[2] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 247–257, New York, NY, USA, 2010. ACM. 1, 24

[3] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power & energy estimation tool. *URL: http://www. drampower. info*, 2012. 22

[4] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society. 25

[5] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In

Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two, IJCAI'11, pages 1237–1242. AAAI Press, 2011. 1

[6] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *Artificial Neural Networks and Machine Learning–ICANN 2014*, pages 281–290. Springer, 2014. 1

[7] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Scene parsing with multiscale feature learning, purity trees, and optimal covers. *arXiv preprint arXiv:1202.2160*, 2012. 19

[8] C. Farabet, C. Poulet, J.Y. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 32–37, Aug 2009. 1, 2, 24

[9] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989. 1

[10] Xilinx Inc. Xilinx power estimator user guide. UG440 (v2015.3), Septembet 30,2015. 22

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. 21

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. vi, 1, 2, 8, 9, 15, 17, 19

[13] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014. 1

[14] M. Peemen, A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19, Oct 2013. 2, 19, 24, 25

[15] Maurice Peemen, Bart Mesman, and Henk Corporaal. Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 169–174, San Jose, CA, USA, 2015. EDA Consortium. 25

[16] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, 2016. ACM. vii, 22, 23

[17] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. 1

[18] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H.P. Graf. A massively parallel coprocessor for convolutional neural networks. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 53–60, July 2009. 2, 24

[19] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. 19, 22

[20] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM. vi, 2, 5, 7, 8, 15, 18, 19, 20, 21, 25